5 Reusable React Hook Recipes for Everyday Development

gudoop.com/2025/07/17/5-reusable-react-hook-recipes-for-everyday-development/

Gudoop

Introduction

If you've been building React apps for a while, you know how often you repeat the same logic—debouncing an input, syncing data with localStorage, tracking screen size, or detecting outside clicks.

That's where **custom React hooks** shine.

Hooks allow you to abstract common logic into a **reusable function** that can be plugged into any component. In this post, we'll go through **5 useful custom React hook recipes** you can copy and use in your next project.

No fluff. Just practical hooks that solve real problems.

1. useDebounce – Delay User Input Updates

Why?

You don't want to hit an API on every keystroke in a search field. Instead, you want to **wait until the user stops typing**.

```
// useDebounced.js
import { useEffect, useState } from "react";

export function useDebounce(value, delay = 500) {
  const [debounced, setDebounced] = useState(value);

  useEffect(() => {
    const handler = setTimeout(() => setDebounced(value), delay);
    return () => clearTimeout(handler);
  }, [value, delay]);

  return debounced;
}
```

Real-World Example

```
const [search, setSearch] = useState("");
const debouncedSearch = useDebounce(search, 300);

useEffect(() => {
  if (debouncedSearch) {
    fetch(`/api/products?q=${debouncedSearch}`)
      .then(res => res.json())
      .then(data => console.log(data));
  }
}, [debouncedSearch]);

return <input value={search} onChange={e => setSearch(e.target.value)} />;
```

2. useLocalStorage – Persist Data Between Page Loads

Why?

Sometimes you want to store user preferences (like theme) or form inputs and keep them available even after a refresh.

```
// useLocalStorage.js
import { useEffect, useState } from "react";

export function useLocalStorage(key, initialValue) {
  const [value, setValue] = useState(() => {
    const stored = localStorage.getItem(key);
    return stored ? JSON.parse(stored) : initialValue;
  });

  useEffect(() => {
    localStorage.setItem(key, JSON.stringify(value));
  }, [key, value]);

  return [value, setValue];
}
```

Real-World Example

```
const [theme, setTheme] = useLocalStorage("theme", "light");

return (
  <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>
    Switch to {theme === "light" ? "dark" : "light"} mode
  </button>
);
```

3. useClickOutside – Detect Outside Clicks

Why?

You want to **close a dropdown** or modal when the user clicks outside of it. It's a common UX pattern.

```
// useClickOutside.js
import { useEffect } from "react";

export function useClickOutside(ref, handler) {
  useEffect(() => {
    function listener(e) {
      if (!ref.current || ref.current.contains(e.target)) return;
      handler(e);
    }
    document.addEventListener("mousedown", listener);
    return () => document.removeEventListener("mousedown", listener);
  }, [ref, handler]);
}
```

Real-World Example

```
const ref = useRef();
const [open, setOpen] = useState(false);

useClickOutside(ref, () => setOpen(false));

return (
  <div ref={ref}>
    <button onClick={() => setOpen(o => !o)}>Toggle</button>
    {open && <div className="dropdown">Dropdown content</div>}
  </div>
);
```

4. useWindowSize – Track Screen Dimensions

Why?

Responsive design often depends on the screen size. Instead of using a CSS media query, you can handle it in JS.

```
// useWindowSize.js
import { useState, useEffect } from "react";

export function useWindowSize() {
  const [size, setSize] = useState({
    width: window.innerWidth,
    height: window.innerHeight,
  });

  useEffect(() => {
    const handleResize = () =>
      setSize({ width: window.innerWidth, height: window.innerHeight });

    window.addEventListener("resize", handleResize);
    return () => window.removeEventListener("resize", handleResize);
  }, []);

  return size;
}
```

Real-World Example

```
const { width } = useWindowSize();

return (
  <div>
    {width < 768 ? <MobileMenu /> : <DesktopMenu />}
  </div>
);
```

5. usePrevious – Track the Previous Value

Why?

Need to compare a prop or state's previous value to the current one? This hook stores the previous state value between renders.

```
// usePrevious.js
import { useRef, useEffect } from "react";

export function usePrevious(value) {
  const ref = useRef();
  useEffect(() => {
    ref.current = value;
  }, [value]);
  return ref.current;
}
```

Real-World Example

```
const [count, setCount] = useState(0);
const prevCount = usePrevious(count);

return (
  <div>
    <p>Now: {count}</p>
    <p>Before: {prevCount}</p>
    <button onClick={() => setCount(c => c + 1)}>Increment</button>
  </div>
);
```

Conclusion

These 5 reusable React hook recipes are game-changers for day-to-day development. Not only do they **simplify your code**, but they also promote **clean architecture** and **reusability** — especially when working on large applications.

Rather than writing the same boilerplate again and again, create a custom hook once and reuse it across your app or even across projects.

> Pro Tip: Put these hooks into a shared `hooks/` folder and create your personal React developer toolkit!